A Knowledge-Based Method for Inferring Semantic Concepts from Graphical Models of Real-time Systems

KEVIN L. MILLS
National Institute of Standards and Technology, Gaithersburg, Maryland 20899

HASSAN GOMAA
George Mason University, Fairfax, Virginia 22030

Abstract. Designers of software for real-time systems often use models, in the form of data flow/control flow diagrams, to express system behavior in a graphical notation that can be understood easily by users and by programmers, and from which designers can generate a software architecture. The research described in this paper is motivated by the desire to provide an automated designer's assistant that can generate software architectures for real-time systems directly from models expressed as data flow/control flow diagrams. To achieve this goal, an automated designer's assistant must be capable of interpreting flow diagrams in semantic, rather than syntactic, terms. Unfortunately, flow diagrams, with a limited number of syntactic elements, are seldom expressive enough to depict the semantic concepts that a designer intends; instead, most design methods that include flow diagrams allow the designer to augment those diagrams with a textual description to express semantic information. This paper proposes a knowledge-based approach capable of inferring the presence of semantic concepts from data flow/control flow diagrams. To illustrate the approach, the paper specifies a knowledge-based graphical model for data flow/control flow diagrams used in the context of a specific modeling method for real-time systems, Concurrent Object-based Real-time Analysis (COBRA). In addition, the paper applies the approach to the design of software for an automated cruise-control system and provides an evaluation of the approach based upon results from four case studies. For the case studies, the knowledge-based graphical model recognized, automatically and correctly, the existence of 86% of all COBRA semantic concepts within the flow diagrams. Varying degrees of human assistance were used to correctly identify the remaining semantic concepts within the diagrams: in two percent of the cases the knowledge-based graphical model reached tentative classifications that a designer was asked to confirm or override; in four percent of the cases a designer was asked to provide additional information before a concept was classified; in the remaining eight percent of the cases, all involving terminators, the designer was asked to identify the semantic concept.

Keywords: software design methods, concept classification systems, knowledge-based software engineering, semantic data modeling, real-time systems

1. Introduction

Designers of software for real-time systems often use models, in the form of data flow/control flow diagrams, to express system behavior in a graphical notation that can be understood easily by users and by programmers, and from which designers can generate a software architecture. The research described in this paper is motivated by the desire to provide an automated designer's assistant that can generate software architectures for real-time systems directly from a graphical model expressed as data flow/control flow diagrams (Mills, 1996; Mills and Gomaa, 1996). To achieve this goal, an automated designer's assistant must be capable of interpreting a flow diagram in semantic, rather than syntactic, terms. Unfortunately, flow diagrams, with a limited number of syntactic elements and a feeble underlying semantics, are seldom expressive enough to depict the semantic concepts that a designer intends; instead, most design methods that include flow diagrams allow the designer to augment those diagrams with a textual description to express semantic information. Such semantic information might be included directly in a flow diagram model by asking a human designer to tag the syntactic elements of a flow diagram with semantic labels appropriate for concepts within a particular design method. Such a manual strategy would prove tedious, error-prone, and costly. More elegant methods should be possible to enable an automated designer's assistant to infer the presence of semantic concepts from a flow diagram. This paper describes such a method, knowledge-based graphical models.

Knowledge-based graphical models encompass a class of models where the syntactic elements of a graphical notation are connected, through a knowledge base, to concepts in an underlying semantic model appropriate to a particular design method. The knowledge base serves three purposes: (1) it enables many syntactic elements in the graphical model to be classified automatically as semantic concepts from the design method, (2) it guides the automatic elicitation of information required for a complete semantic specification of a design, and (3) it tests assertions about the presence of semantic concepts in the graphical model. To illustrate the approach, Section 5 explains how knowledge-based graphical models can overcome the semantic gap between the syntax of a graphical notation and the semantic concepts embodied within a design method. The explanation is presented through the development of a knowledge-based graphical model for a specific modeling method for real-time systems, COBRA, or Concurrent,

Object-based Real-time Analysis (Gomaa, 1993). A human designer can readily transform COBRA models into concurrent designs using an accompanying software design method known as COncurrent Design Approach to Real-Time Systems, CODARTS. After Section 5 presents our approach for modeling COBRA, Section 6 describes a means of implementing knowledge-based graphical models. First, however, Section 4 introduces the use of graphical models for system analysis and software design, and provides an overview of both COBRA and CODARTS. Section 7 applies the knowledge-based graphical model of COBRA to design software for an automated cruise-control system. Section 8 uses results from four case studies to evaluate the knowledge-based graphical model for COBRA. The concluding section provides a summary of the paper and suggests some directions for future work. First, though, Section 2 compares briefly our approach to automated software design with some related work from other researchers, and Section 3 places the ideas described in this paper within the context of an automated designer's assistant.

2. Comparison With Related Work

In order to facilitate traceability between detailed designs and behavioral models, to expedite the design process, and to improve the quality of software designs, a number of researchers explore automated assistants to help designers generate software architectures, typically from data flow diagrams. Most of these automated design assistants (Boloix, Sorenson, and Tremblay, 1992; Karimi and Konsynski, 1988; Tsai and Ridge, 1988) generate software architectures, in the form of structure charts, from a low-level mechanical analysis of the syntactic elements comprising flow diagrams, that is, data flows, processes, data stores, and terminators. A significant problem arises because data flow diagrams, even when expanded to include control flows and control transformations, provide a rather restricted set of syntactic elements from which to model the semantics of a particular design method. Design assistants based on low-level analyses fail to achieve the rich semantic interpretation that a human designer gives to flow diagrams; thus, these approaches prove ineffective. Another, more promising, approach (Lor and Berry, 1991) generates designs for concurrent systems from a conceptually richer model that augments data flow diagrams with a specification of allowable parallelism. Unfortunately, Lor's approach succeeds by forcing the designer to use semantic constructs that encode a particular concurrent design within the system model. Unlike the approach of Lor and

Berry, our approach does not require the designer to restrict the concurrency present in the system model in order to generate a concurrent software design.

3. Overview of Concurrent Designer's Assistant

Figure 1 illustrates one view of the architecture for a concurrent designer's assistant, CODA, that embodies our approach. Given a data flow/control flow diagram model and a description of the intended target environment, along with any design guidelines, CODA largely automates the process of generating a concurrent design, consisting of a software architecture, initial specifications for tasks and for information hiding modules, and a consistency analysis of the resulting design. Conceptually, CODA consists of two main components: a model analyzer and a design generator. The model analyzer converts a syntactically described flow diagram into a flow diagram annotated with semantic concepts from a specific modeling method, Concurrent Object-Based Real-time Analysis, COBRA (see Section 4.1). The design generator uses design knowledge from a specific design method, COncurrent Design Approach for Real-Time Systems (see Section 4.2), to transform an annotated flow diagram into a concurrent design. The current paper focuses on the model analyzer; the design generator is discussed elsewhere (Mills, 1996; Mills and Gomaa, 1996). The model analyzer consists of a meta-model that encodes the *static portions of the knowledge-based graphical model for COBRA* and three rule sets that encode the *inference portions of the knowledge-based graphical model for COBRA*, including knowledge about classifying concepts, checking axioms, and eliciting missing information (these are explained in Section 5, while Section 6 describes a means of implementing these ideas). In practical application (see Section 7), CODA largely automates the transformation of COBRA models of real-time systems into concurrent software designs, guided by constraints describing the intended target environment. The overall design, application, and operation of CODA are described elsewhere (Mills, 1996; Mills and Gomaa 1996); this paper concentrates on the ideas underlying the model analyzer.

4. Graphical Models for Software Analysis and Design

Numerous software design methods include an initial step where a system is analyzed and modeled using a graphical notation and a textual description language. In subsequent steps, the graphical model and accompanying textual description are transformed into a software architecture that can guide further development of a software solution. Examples of such design

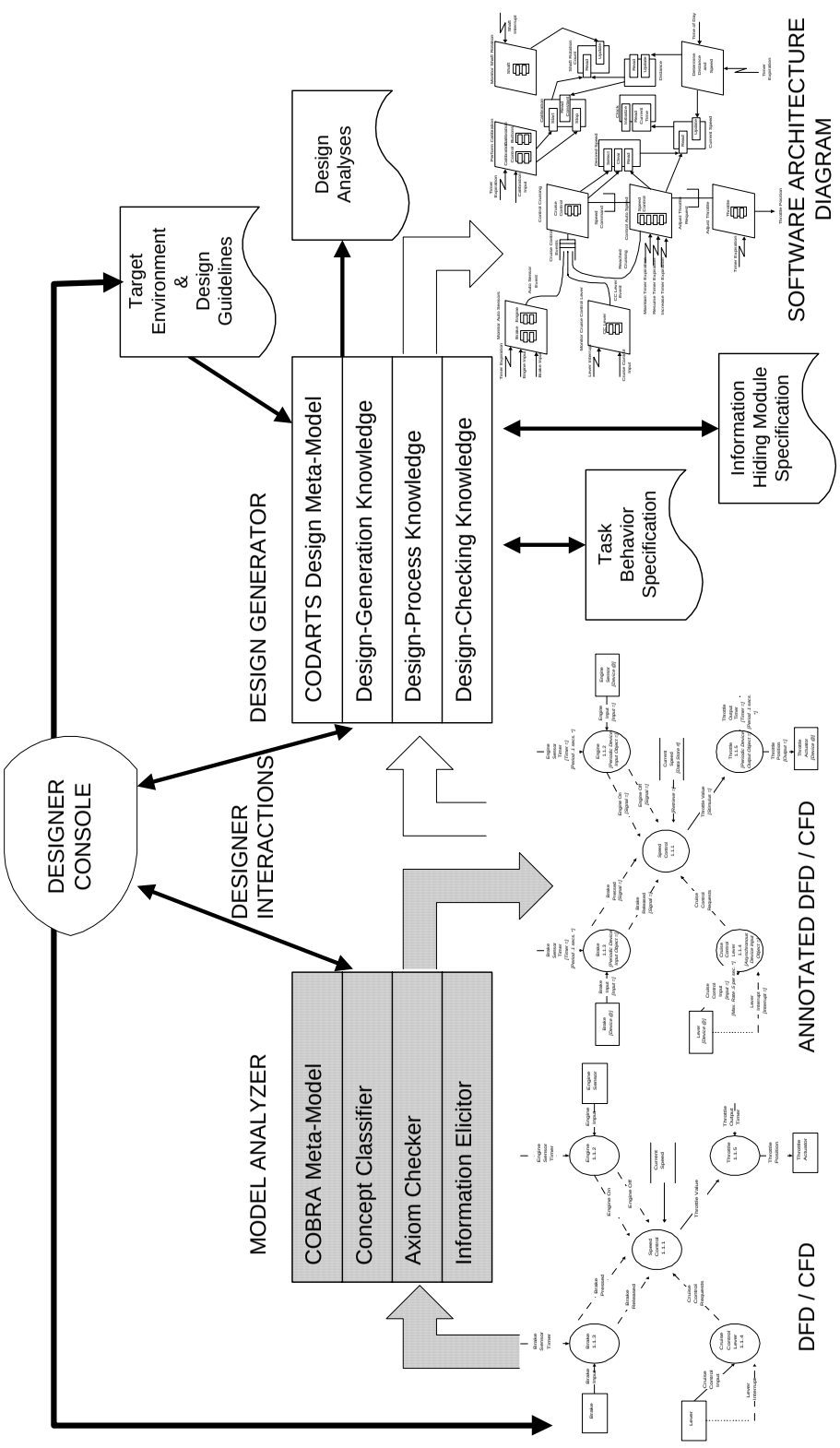

Figure 1. Conceptual Architecture for COncurrent Designer's Assistant, CODA

methods include: structured analysis and design (DeMarco, 1978; Yourdon and Constantine, 1979) ; Jackson System Development (Jackson, 1983); real-time structured analysis and design (Ward and Mellor, 1985; Mellor and Ward, 1986; Hatley and Pirbhai, 1988); and object-oriented analysis and design (Booch, 1986; Booch, 1991; Coad and Yourdon, 1991; Rumbaugh et al, 1991; Shlaer and Mellor, 1992). To move effectively from a graphical model of a problem to a software architecture, a designer uses information from the textual description to interpret the elements of the graphical model with a semantic significance that is not always readily apparent from the graphical notation alone. This semantic gap between a designer's intent and the graphical model inhibits the ability of researchers to construct automated design assistants that operate at the same semantic level as a human designer.

This paper explains, in Section 5, how knowledge-based graphical models can overcome the semantic gap between the syntax of a graphical notation and the semantic concepts embodied within a design method for real-time systems. To illustrate our approach, we use a specific design method that consists of a modeling method for real-time systems, COBRA, and an accompanying software design method known as CODARTS. Before presenting our approach, the COBRA and CODARTS methods are explained briefly below.

4.1 Concurrent Object Based Real-time Analysis

COBRA comprises an analysis method and a set of semantic conventions for modeling a system using graphical notations, as shown in Figure 2, from Real-Time Structured Analysis (RTSA) (Ward and Mellor, 1985). The results from applying COBRA to a real-time system include two main models: (1) an environmental model, shown with a system context diagram that distinguishes a system under design from its environment and (2) a behavioral model, illustrated through a set of hierarchically organized data flow/control flow diagrams and state transition diagrams, accompanied by a textual description of the elements on the diagrams. The behavioral model is augmented with event sequence diagrams that depict the reaction of the system to selected scenarios of external events.

While the graphical depiction of a COBRA model is identical to a RTSA model, during system analysis and modeling, the COBRA method guides a designer to place semantic interpretation upon the syntactic elements of a COBRA model. Figure 3 illustrates the main

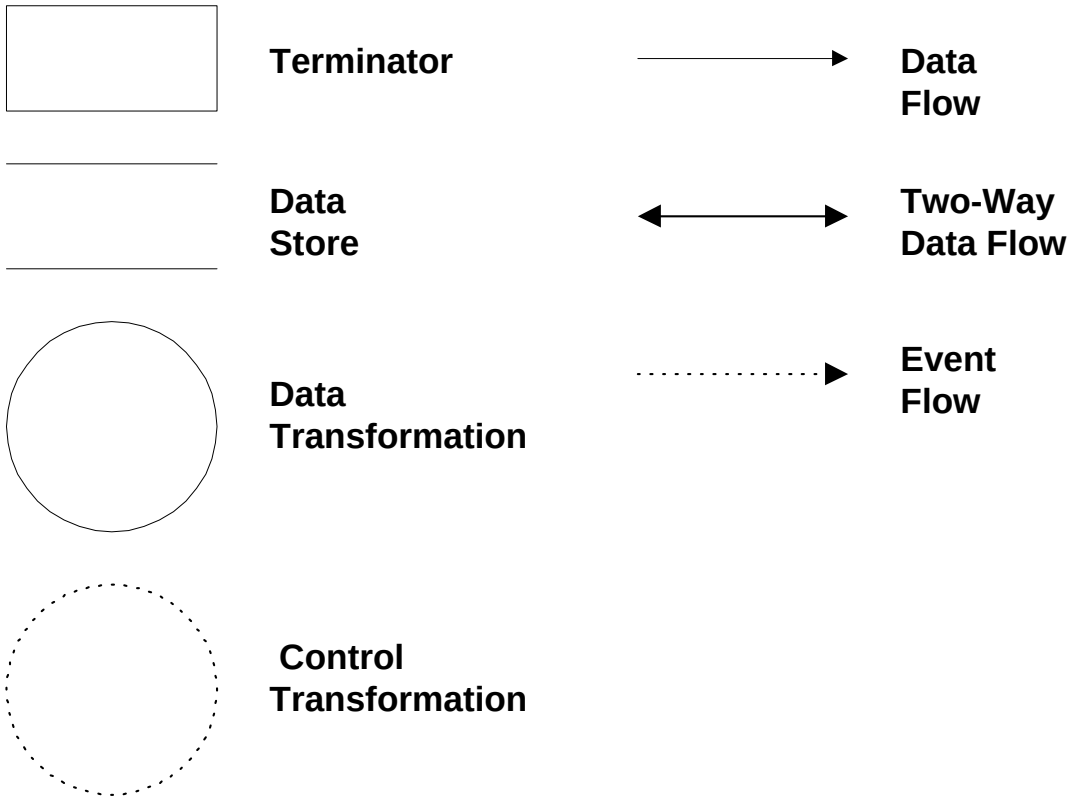

Figure 2. Syntactic Elements for Composing RTSA Data Flow/control Flow Diagrams

semantic concepts included within the COBRA method. The semantic concepts are divided into five main classes: terminators, objects, functions, event flows, and data flows.

COBRA distinguishes three types of terminators: devices, which correspond to real hardware with which the system interacts; user roles, which correspond to a type of user that might interact with the system through commands from a terminal; and external subsystems, which represent other systems or subsystems that communicate with the system under design. Data flows from and to terminators represent inputs and outputs, respectively. Event flows from devices denote interrupts.

COBRA uses data transformations and control transformations in RTSA to represent objects and functions. Edge data transformations can be distinguished based on the type of terminator to which they connect. For example, data transformations that connect to devices are either: input, output, or input/output device objects, depending on the direction of data flow.

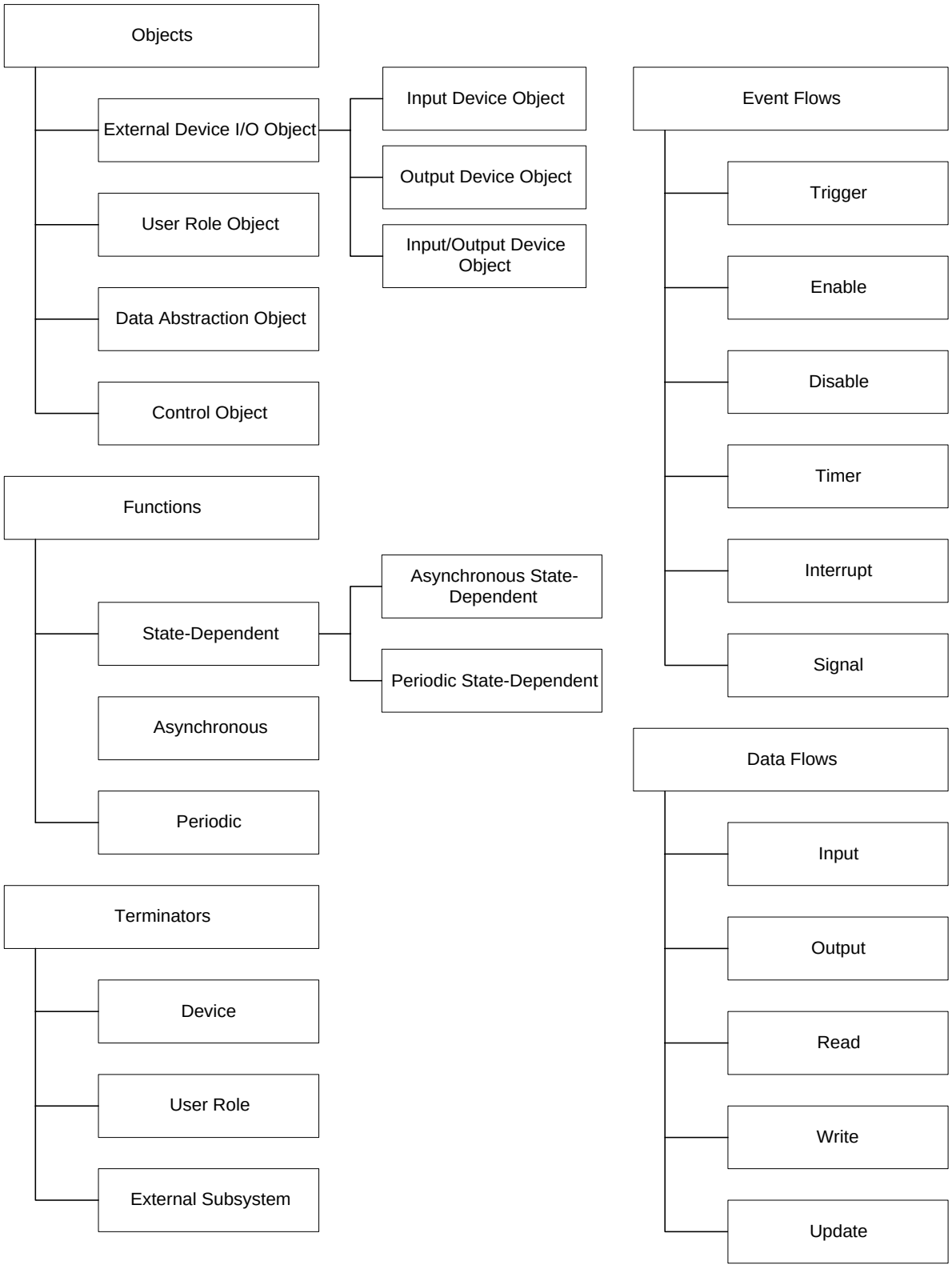

Figure 3. Semantic Concepts From COBRA Behavioral Modeling Method

Similarly, COBRA data abstraction objects consist of RTSA data stores and appropriately connected data transformations.

COBRA uses internal data transformations (i.e., those that do not connect to terminators or to data stores) to represent various types of function, where each function must be activated by receiving some data flow or event flow: For example, periodic functions are activated cyclically by a timer event.

COBRA views each RTSA control transformation as a control object with an embedded state-transition diagram. Control objects receive event flows and, depending on the internal state of the embedded state-transition diagram, emit several types of event flows (e.g., enable, disable, or trigger) that control the execution of data transformations (specifically, state-dependent functions). Control objects can also emit signals that indicate the occurrence of some condition.

4.2 COncurrent Design Approach for Real-time Systems

Once a COBRA model exists, CODARTS (Gomaa, 1993) provides four steps for transforming that model into a concurrent design: (1) Task Structuring, (2) Task Interface Definition, (3) Module Structuring, and (4) Task and Module Integration. First, CODARTS task structuring criteria assist a designer in examining a COBRA model to identify concurrent tasks. The task structuring criteria, consisting of a set of heuristics derived from experience obtained in the design of concurrent systems, can be grouped into four categories: input/output task structuring criteria, internal task structuring criteria, task cohesion criteria, and task priority criteria. In a given design, a task may exhibit several criteria and many tasks may exhibit the same criteria. The input/output and internal task structuring criteria identify tasks based upon how and when a task is activated: periodically based on the need to poll a device or to perform a calculation; asynchronously based on an external device interrupt or on an internal event. The task cohesion criteria help a designer to identify COBRA objects and functions that can be combined together within the same task, e.g., because a set of operations: (1) must be performed sequentially (sequential cohesion), (2) can be performed with the same period (temporal cohesion), or (3) perform a set of closely related functions (functional cohesion). The task priority criteria prevent a designer from combining tasks that might need to execute at substantially differing priorities.

As a second step, CODARTS provides guidelines for defining inter-task interfaces. Once tasks are defined, data and event flows from a COBRA model can be mapped to inter-task signals or to tightly-coupled or loosely-coupled messages, depending on the synchronization requirements between specific pairs of tasks. As a third step, CODARTS includes criteria, based on information hiding, to help a designer identify modules from the objects and functions in a COBRA model. The CODARTS module structuring criteria form modules to hide the details of: device characteristics, data structures, state-transition diagrams, and algorithms. As a final step, once both the task and module views of a concurrent design exist, CODARTS provides guidelines to help a designer relate the independent views into a single, consistent design. Each task represents a separate thread of control, activated by some event: such as an interrupt, a timer, or a message arrival. Each module provides operations that can be accessed by the tasks in a design. CODARTS helps a designer to establish the control flow from events to tasks and then on to operations within modules.

5. A Knowledge-Based Graphical Model for COBRA

A knowledge-based graphical model for COBRA requires two major components: a model of COBRA models, called a COBRA meta-model, and a knowledge base. The COBRA meta-model defines the semantic concepts included in COBRA and describes the relationships permitted and prohibited among those concepts. The knowledge base links syntactic elements from COBRA graphical models to concepts within the COBRA meta-model, and defines any additional information that must be supplied to satisfy the semantic requirements of a COBRA concept. The COBRA meta-model consists of a concept taxonomy and concept axioms, discussed below in Sections 5.1 and 5.2, respectively. The COBRA knowledge base, consisting of concept classification rules and information elicitation rules, is described in Section 5.3. A subsequent section, Section 6, outlines how the knowledge-based graphical model for COBRA can be implemented within an automated designer's assistant.

5.1 Concept Taxonomy

A major aspect of the COBRA meta-model includes a concept taxonomy (Fikes and Kehler, 1985; Lim and Cherkassky, 1992), where each child concept specializes, using an **is-a** relationship, its parent concept. Figure 4 shows a partial concept taxonomy for the COBRA semantic meta-model. A complete specification of the taxonomy can be found elsewhere (Mills,

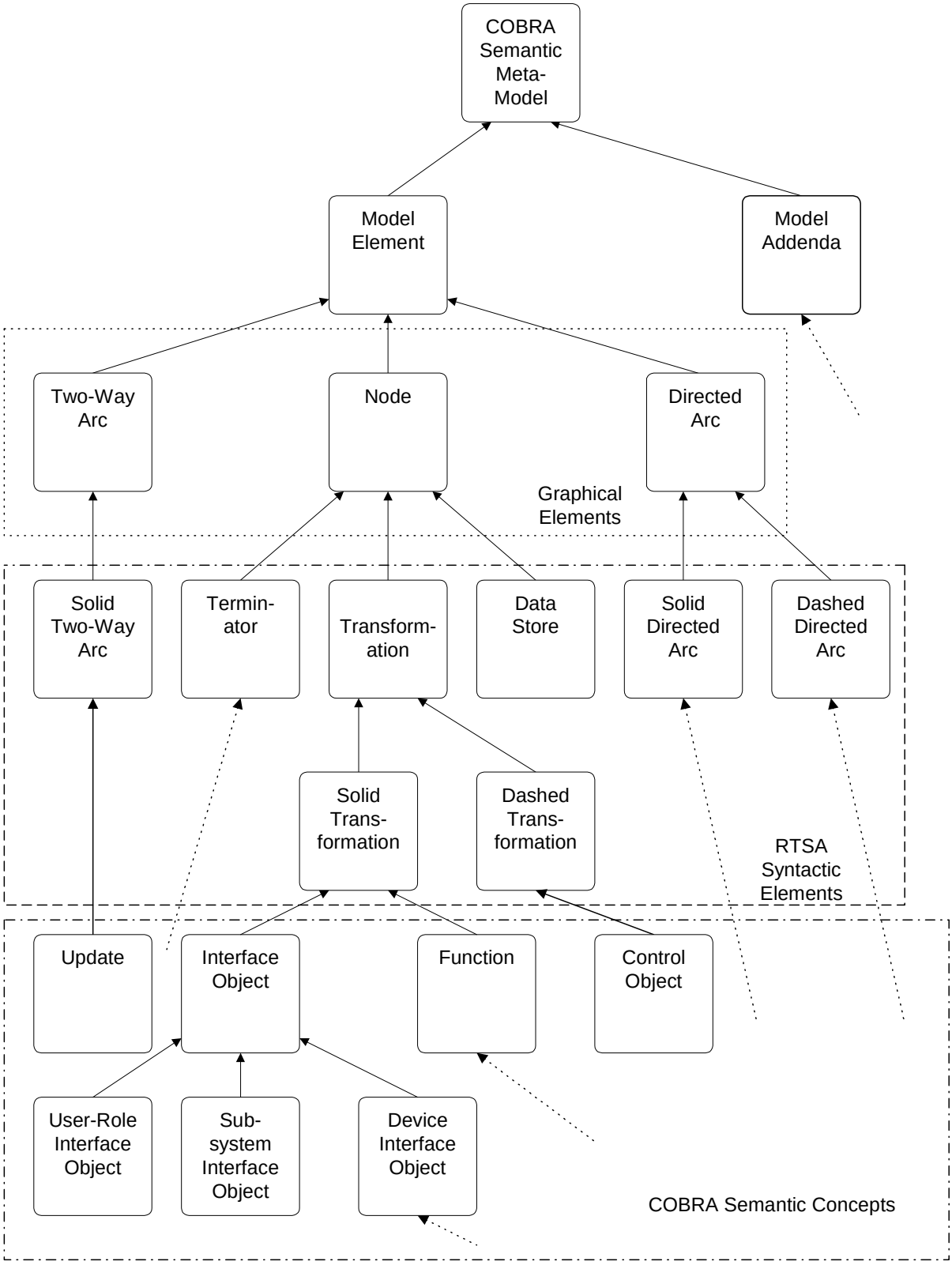

Figure 4. Partial Concept Taxonomy for the COBRA Meta-Model

1996). In Figure 4, each rounded rectangle denotes a semantic concept; Node, for example. Each directed arc, drawn from a child concept to the parent concept(s), represents an **is-a** relationship. For example, the concept Data Store in Figure 4 **is-a** Node. The six concepts appearing on Figure 4 with a dashed arrow pointing to them, for example Terminator, indicate that they are parents to additional child concepts, not shown on the figure.

The concept taxonomy begins with a first-level division of concepts into model elements and model addenda. Model elements represent components of data flow/control flow diagrams, while model addenda represent additional information that, while not depicted on data flow/control flow diagrams, is needed to make design decisions. Model elements are discussed further below.

The specialization of the concept Model Element consists of three layers: (1) the first layer contains the graphical elements that compose directed graphs: nodes, directed arcs, and two-way arcs; (2) the second layer contains the syntactic elements of a RTSA data flow/control flow diagram; and (3) the third layer contains the classification hierarchy for semantic concepts used in COBRA. Figure 4 shows where each of the three layers falls within the concept taxonomy. This layered taxonomy allows the presence of semantic concepts in a given COBRA model to be inferred from the syntactic elements of RTSA data flow/control flow diagrams. Using this concept taxonomy, data flow/control flow diagrams can be denoted with the RTSA syntactic elements. Further specialization of the taxonomy allows additional COBRA semantic concepts to be identified.

Two important sub-trees of the taxonomy distinguish solid (data) transformations as various types of device interface objects and functions. The taxonomy enables device interface objects to be categorized based on the characteristics, or requirements, of the device for which they provide an interface. For example, device interface objects might be input-only or output-only. Device interface objects can also be classified based on the impetus for data exchanges with an associated Device. For example, some devices must be periodically checked for state changes, while other devices generate interrupts when a state change occurs. The various types of device interface object, based on the direction and impetus for data exchange, can be combined, using multiple inheritance, to form nine varieties of object, as shown in Figure 5.

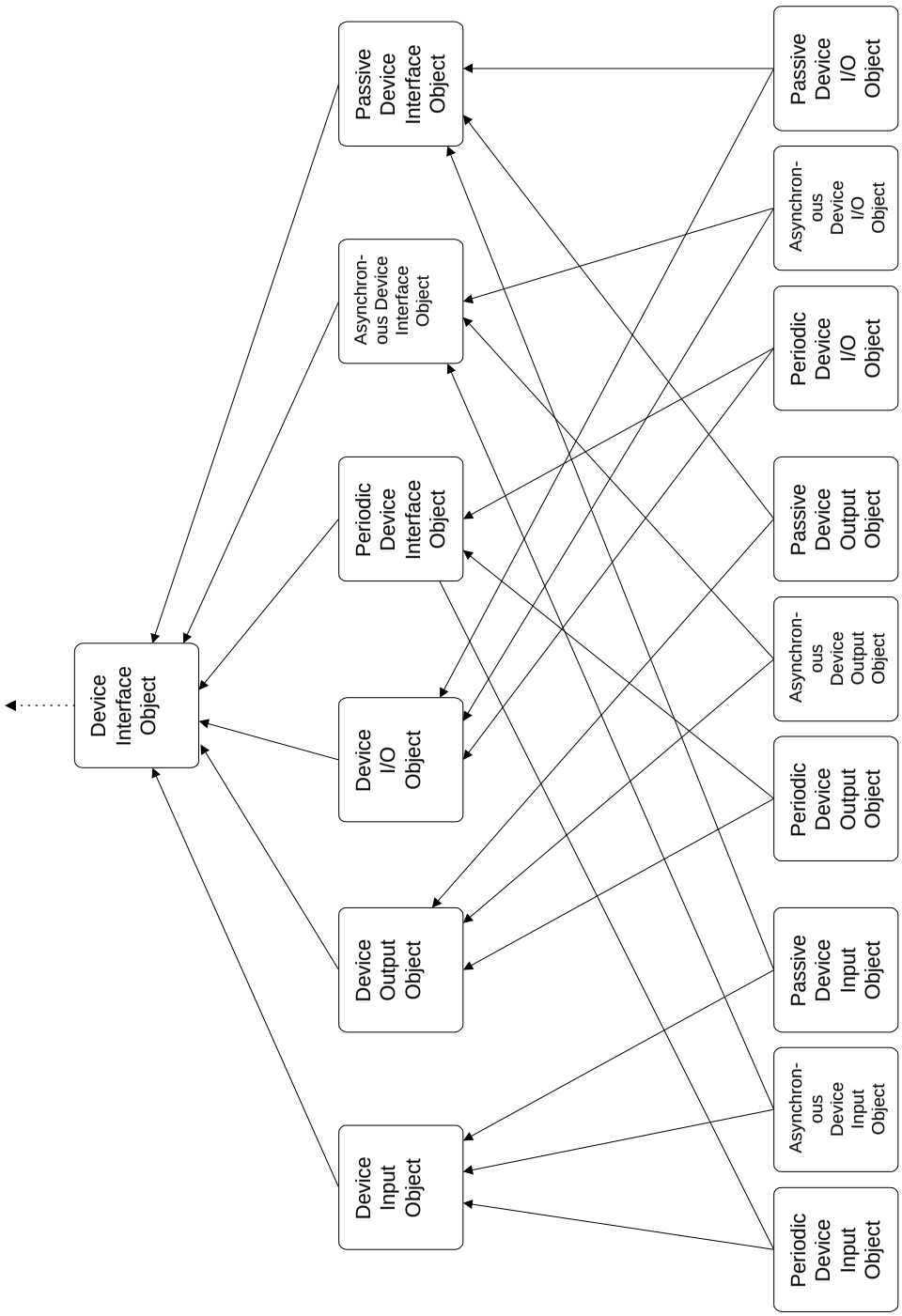

Figure 5. The Device Interface Object Concept Taxonomy

Figure 6 illustrates how data transformations representing functions can be classified. The main division identifies functions as either state-dependent or state-independent. State-dependent functions include internal data transformations that are controlled by the finite-state diagram embedded in a control object. For example, a function, triggered by a control object, might be required to finish its processing at a state-transition. Other functions might be enabled and disabled by a control object. State-independent functions include those internal data transformations that are not managed by control objects. For example, some functions might execute periodically, while others execute on demand.

5.2 Concept Axioms

In the preceding section, semantic concepts composing the COBRA taxonomy were discussed in an informal manner in order to provide the reader with an intuitive understanding. More formal definitions are needed to facilitate automated support: (1) for verifying assertions that instances of a concept are indeed valid instances of the concept and (2) for evaluating whether or not a concept is a leaf in the taxonomy. The COBRA meta-model addresses these requirements by providing axioms, expressed using predicate logic (Genesereth and Ginsberg, 1985), for each semantic concept within the taxonomy, and by allowing axioms from more general concepts to be inherited by more specialized concepts. This section explains the main principles of concept axioms and inheritance. A full specification of the axioms applicable for each concept in the COBRA meta-model can be found elsewhere (Mills, 1996). Each semantic concept within the taxonomy can be constrained by a set of zero or more axioms, where each axiom consists of an axiom name and an axiom body. For example, consider the following axiom that applies to any instance of the concept Periodic Device Interface Object.

Axiom: **One, And Only One, Timer**
Let T be the set of all Timers whose sink is the given Periodic Device Interface Object. The cardinality of T must be one.

The axiom name, **One, And Only One, Timer**, is shown in **boldface** type and the axiom body is shown in *italics*.

Any instance of the concept Periodic Device Interface Object must also meet another axiom, as follows.

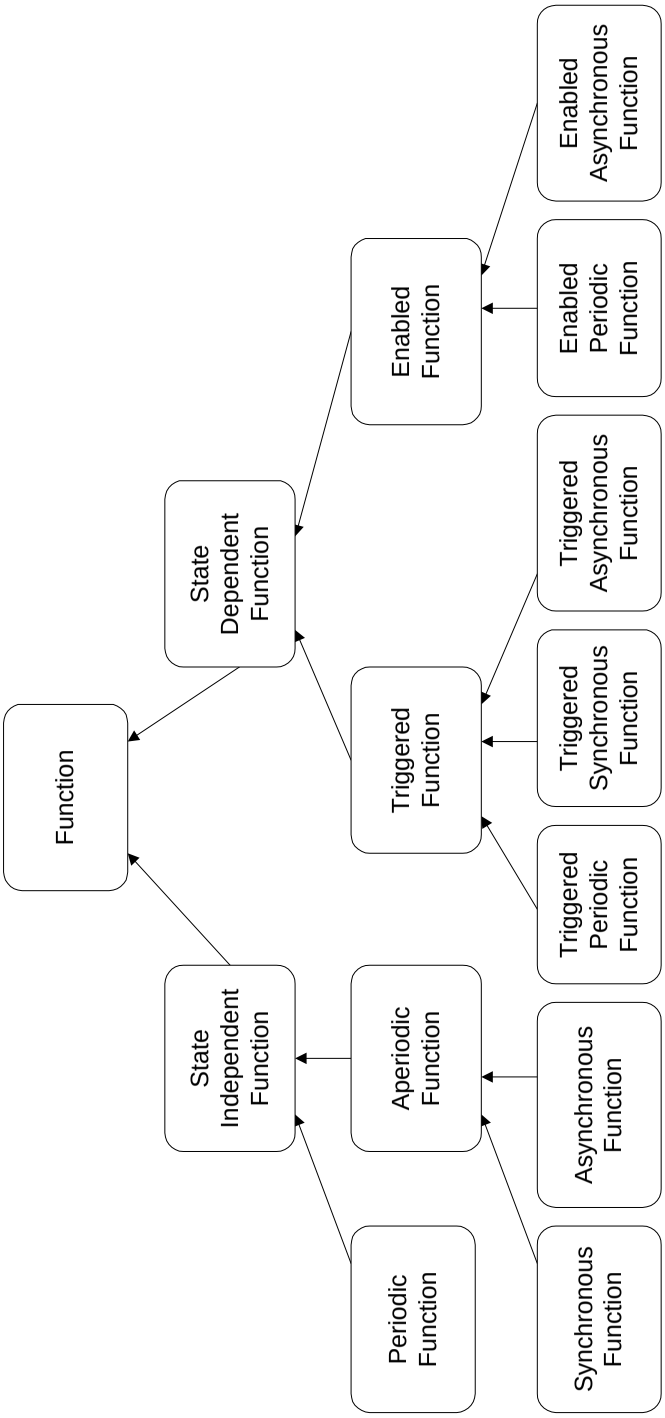

Figure 6. The Function Concept Taxonomy

Axiom: **No Interrupt**
There must not exist an Interrupt whose sink is the given Periodic Device Interface Object.

These two axioms, **One, And Only One, Timer** and **No Interrupt**, must be satisfied by any instance of a concept that includes the concept Periodic Device Interface Object in its inheritance path along the concept taxonomy. Since Periodic Device Interface Object is not a leaf concept, these axioms would be inherited by any of three child concepts: Periodic Device Input Object, Periodic Device Output Object, and Periodic Device I/O Object. The principles of axiom inheritance can best be illustrated through an example.

Consider the axioms that apply to the leaf concept Periodic Device Input Object, as shown in Figure 7. Figure 7 depicts a slice through the concept taxonomy, beginning with the abstract concept Specification Element and ending with the leaf concept Periodic Device Input Object. Concept inheritance is shown, as before, with a directed arc pointing from a child concept to its parent concept(s). Now, for each concept, the names of any associated axioms are shown within the rectangle, below the concept name, following the label Axioms. In Figure 7, the concept Periodic Device Input Object, even though it contains no axioms of its own, must satisfy the following axioms based on inheritance.

Node:	**Name Required**
Node:	**Name "System" Reserved**
Node:	**Distinct Name**
Transformation:	**Distinct Number**
Transformation:	**At Least One Input**
Transformation:	**At Least One Output**
Solid Transformation:	**No Redundant Data Flows**
Interface Object:	**Interface To One, And Only One, Terminator**
Device Interface Object:	**Requires Input, Output, Or Interrupt**
Device Input Object:	**Input Only**
Periodic Device Interface Object:	**One, And Only One, Timer**
Periodic Device Interface Object:	**No Interrupt**

Similar groups of axioms can be composed for any concept in the taxonomy. The group of axioms that apply to a given concept in the COBRA meta-model yield a formal definition for the concept. Any graphical element in a RTSA specification that satisfies the applicable axioms for a semantic concept is a valid instance of that concept. Any graphical element in a RTSA

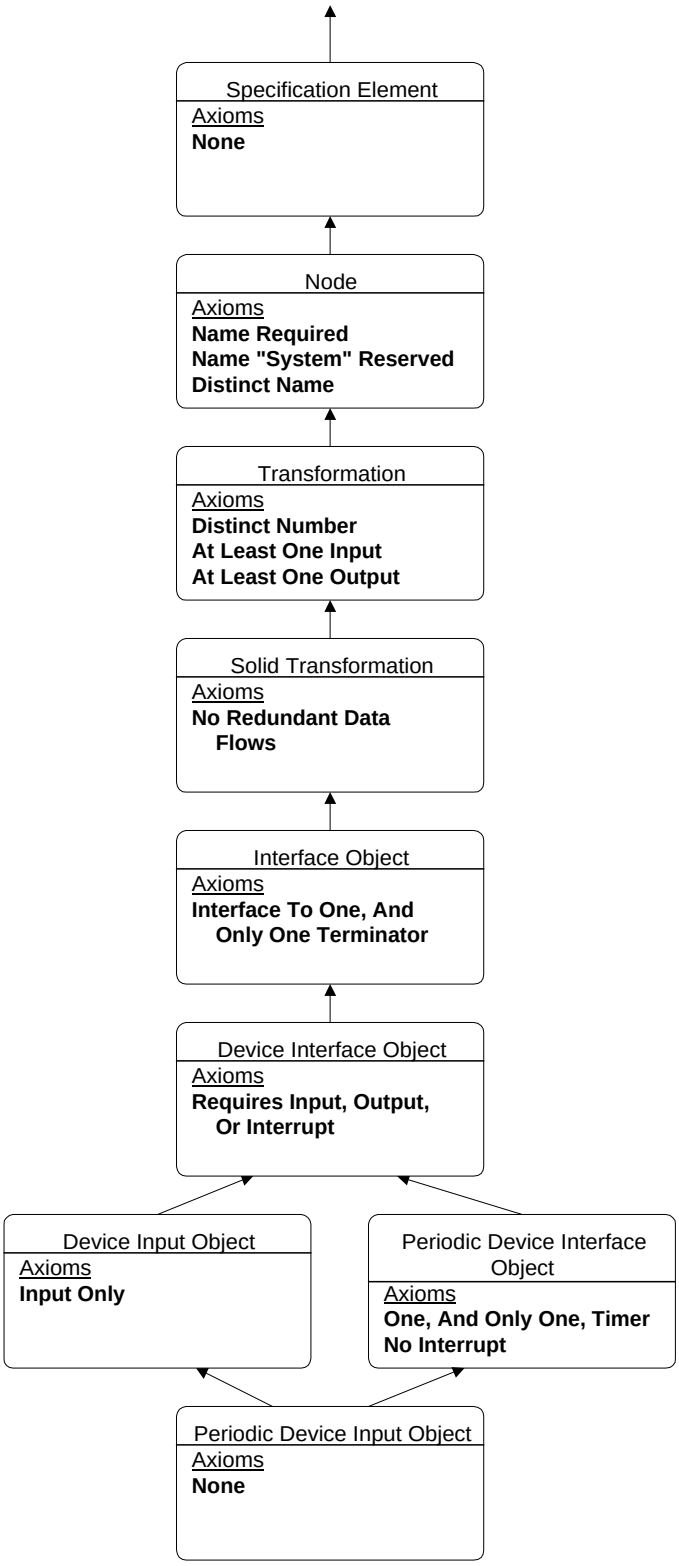

Figure 7. The Concept Taxonomy, Axioms, and Inheritance

specification that is asserted to be a specific, semantic concept and that then does not satisfy the applicable axioms for that concept is not a valid instance of the concept.

5.3 Concept Classification and Information Elicitation

As given to this point, the concept taxonomy specifies the inheritance relationships and general associations between the concepts in COBRA. Two other components of the COBRA semantic meta-model remain to be introduced. The first of these components, classification rules, enables the presence of semantic concepts within COBRA to be inferred from the syntax of RTSA flow diagrams. The second component, information elicitation rules, guides the elicitation of ancillary information, not represented with RTSA syntax, that a designer considers when making design decisions. These two components of the COBRA meta-model are discussed below, beginning with the classification rules.

5.3.1 Classification Rules. Each concept within the concept taxonomy can be augmented with one or more classification rules, where each rule describes a path along the taxonomy from the concept to one of its children. For the COBRA meta-model, each non-leaf concept in the taxonomy, beginning with the RTSA syntactic elements, requires a classification rule for each **is-a** link pointing to the concept from a child. Each classification rule for a COBRA concept is defined using a form of **if-then** rule commonly employed in rule-based expert systems (Hayes-Roth, 1985). The general form for a classification rule follows.

Rule: <u>Classify A Concept</u>

if
a non-leaf concept is recognized and
that concept satisfies the requirements of a more specific concept
then
reclassify the concept as the more specific concept
fi

Here the rule name, <u>Classify A Concept</u>, is shown in <u>underlined</u> text, followed by the definition of the rule itself: **if** antecedent **then** consequent **fi**.

Figure 8 augments Figure 7 with the specific classification rules needed to infer that a solid transformation on a flow diagram is a periodic device input object. In the figure, each inheritance arc is annotated with the classification rule that enables the presence of a more

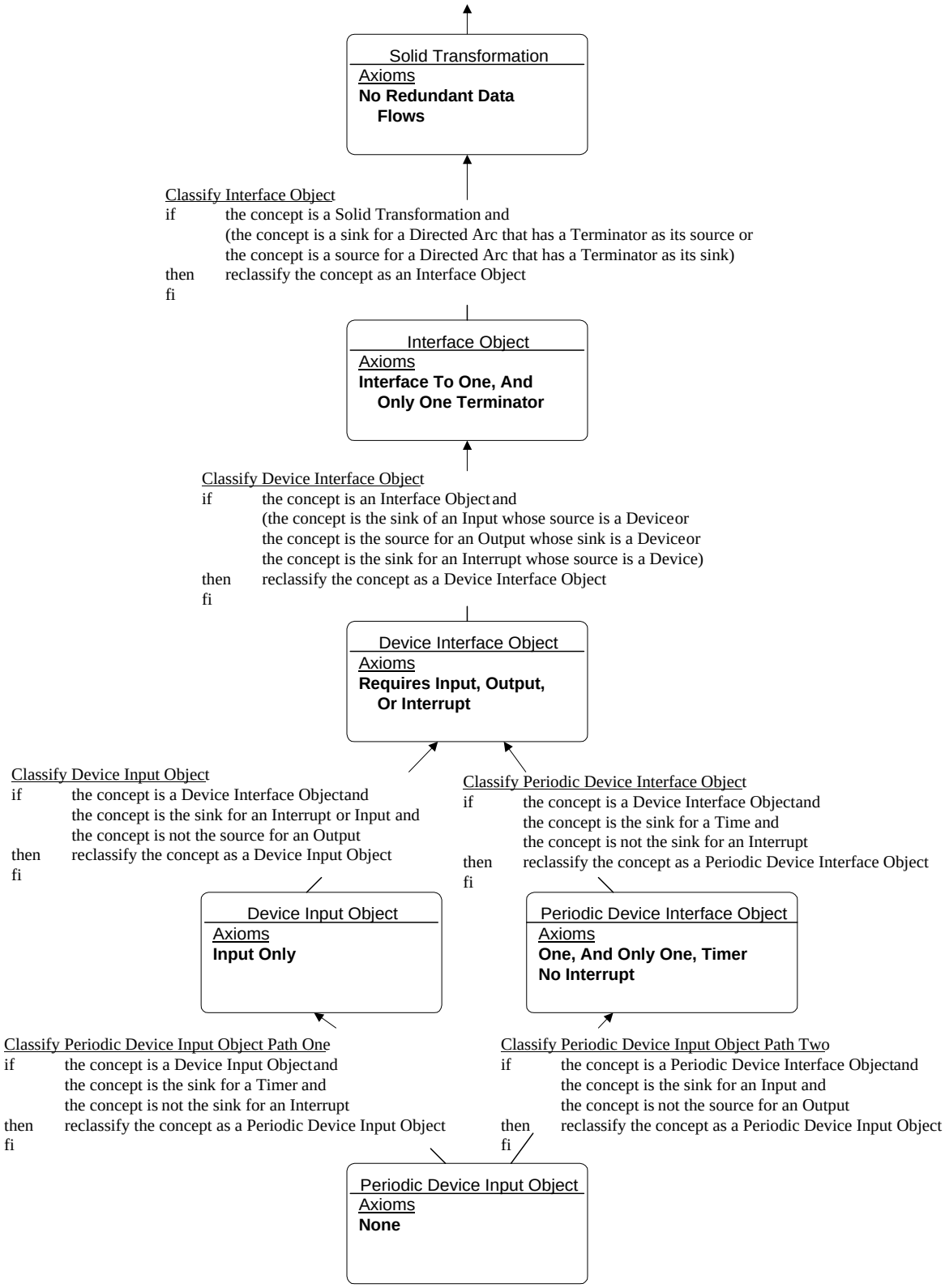

Figure 8. The Concept Taxonomy and Concept Classification Rules

specific concept to be inferred from a more general concept. Note that the inheritance arcs point from the bottom up in the direction of generalization, while the inferences represented by the classification rules flow from the top down.

The first rule, Classify Interface Object, infers that a solid transformation is in fact an interface object, while the second rule, Classify Device Interface Object, infers that an interface object is more particularly a device interface object. Although a device interface object can be classified as one of six more specialized concepts, only two possibilities are represented among the rules illustrated in Figure 8. One rule classifies the transformation as a device input object, while the other classifies it as a periodic device interface object. From either of these rules, a periodic device input object can be inferred. While Figure 8 illustrates the purpose and effect of the concept classification rules by relating them to the concept taxonomy, the process of classification requires that the rules be deployed in an inference network (Michalski, 1980) to form a concept classifier (Brachman and Schmolze, 1989).

5.3.2 Concept Classifier. A COBRA concept classifier must meet a number of objectives. First, classification decisions should be taken automatically, with human input elicited only where no automatic inference can be drawn. This requires that as much classification knowledge as possible be represented within the classification rules. Second, concept classification should assume that only the most minimal information might be provided on the input data flow/control flow diagrams. This means that a flow diagram might be encoded using only the RTSA syntactic concepts (see Figure 2). The concept classifier, then, should be capable of beginning classification from this level of specification. Third, concept classification should make no assumption about the actual starting state of classifications within the input data flow/control flow diagram. This means that a particular flow diagram might contain concepts that are partially classified. The concept classifier, then, should be capable of determining which concepts are represented within a diagram and should begin the classification from the correct point.

The COBRA concept classifier is organized as an inference network, which is a web of classification rules that uses successive rule firings to deduce the presence of COBRA concepts from patterns exhibited among RTSA syntactic elements. The COBRA concept classifier has

Figure 9. A Four-Stage Inference Network for Classifying COBRA Concepts

four stages, as shown in Figure 9. The classifications made in each stage are available to subsequent stages to help in making additional inferences.

Stage one of the network, the Arc Classification stage, accepts a data flow/control flow diagram consisting of RTSA syntactic elements as its input and then attempts to classify all arcs on the diagram, also classifying terminators and transformations, to the extent necessary to classify the arcs. A designer is permitted to classify each terminator; however, unless instructed otherwise, the classifier assumes that each terminator is a device. Unclassified or partially classified specification elements are left for later stages.

Stage two, Transformation Classification, classifies all device interface objects. In addition, stage two, classifies periodic functions and three classes of state-dependent function. No classification decisions made during stage two require interactions with a designer. More subtle classification decisions, sometimes requiring interaction with a designer, are left for the remaining classification stages.

Stage three, Stimulus-Response Classification, classifies each data flow that connects two data transformations (called an Internal Data Flow in Figure 9). In one case the classification rules cannot distinguish a data flow as either a stimulus or a response. This case arises whenever a pair of data transformations is connected by a pair of data flows that point in opposite directions. In this rare case a designer is asked to identify which of the two data flows is the stimulus. Default classification rules exist for cases where the designer cannot help.

Stage four, Ambiguous-Function Classification, examines aperiodic functions, whether triggered by a control object or not, to distinguish those that can execute immediately and quickly from those that might be delayed because an input is unavailable or because the processing time is substantial. In cases where the classification rules cannot make a determination, a designer is asked to supply the additional facts. For example, a designer might be asked whether or not the function completes during a triggering state transition.

5.3.3 Information Elicitation. Certain information required to make design decisions might not be represented directly on a data flow/control flow diagram. For example, cases of mutual exclusion cannot be represented directly on the diagram. In addition, use of the classification rules to automatically classify components from a data flow/control flow diagram might lead to the need for additional information in order to make subsequent design decisions. For example,

if an arc is classified as a timer, then a positive period must be supplied. In general, when this additional information is needed but not found in the flow diagram, a designer must be consulted. The elicitation of the necessary information can be automated to the extent that: (1) the needed information can be identified automatically, (2) a designer can be prompted for the necessary information, and (3) certain consistency checks can be performed on the information supplied by the designer.

6. Implementing Knowledge-Based Graphical Models

The COBRA meta-model forms the basis for implementing an automated analyzer for data flow/control flow diagrams. Such a model analyzer, and the related COBRA meta-model, were implemented as components of a COncurrent Designer's Assistant (CODA) (Mills, 1996). CODA is expressed through an expert-system shell, Version 6.0 of the C Language Integrated Production System (CLIPS), (NASA, 1993) using various knowledge representation techniques, as shown in Table 1.

Table 1. Knowledge Representation for Elements of the COBRA Meta-Model and the CODA Model Analyzer

	Element	**CLIPS Knowledge Representation**
COBRA Meta-Model	Concept	Object Class
	Concept Taxonomy	Class-Inheritance Hierarchy
	Concept Axiom	Class-Query Specification
	Concept Classification Rule	Class-based Expert-System Rule
CODA Model Analyzer	Concept Classification	CLIPS Inference Engine and Modules
	Classification Verification	Class Method and Subclass Checking
	Axiom Verification	Class Method and Class-Query Evaluation
	Information Elicitation	Procedure within a Function

The COBRA meta-model maps directly and naturally to knowledge representations provided by the CLIPS Object-Oriented Language (COOL). COBRA semantic concepts become COOL object classes, while the COBRA concept taxonomy maps onto a COOL class-inheritance hierarchy. Each COBRA concept axiom can then be expressed as the specification for a query against COOL object classes, while each classification rule for a COBRA concept becomes a

class-based expert-system rule in CLIPS. Using this representation for the COBRA meta-model, the CODA model analyzer can perform four functions using additional knowledge representation techniques available within CLIPS/COOL.

The CLIPS inference engine drives the concept classification function of the flow-diagram analyzer by firing expert-system rules that recognize and classify COBRA semantic concepts. Ordering dependencies required by the four-stage inference network (Figure 9) are ensured by encapsulating the expert-system rules corresponding to each stage within a separate CLIPS module and then activating each module in turn. The classification and axiom verification functions of the model analyzer are implemented using class-method invocation. Specifically, one COOL class, Model Element, representing the abstract COBRA concept of the same name, is given two class methods: one, **check-classification**, simply determines whether or not an instance of the class has any subclasses, while the other, **check-axioms**, does nothing. Since every concept in the COBRA concept taxonomy inherits from Model Element, any instance of a concept can be asked, using **check-classification**, whether or not it is a leaf in the taxonomy. Classification verification for an entire flow diagram occurs simply by invoking the **check-classification** method on all class instances. Axiom verification occurs more subtly. Each COOL class representing a COBRA concept inherits from the class Model Element, but refines the class-method **check-axioms** by providing an after method. This after method encapsulates the class-query specifications representing the axioms defined for the COBRA concept represented by the COOL class. When the class-method **check-axioms** is invoked for a class instance, all after methods in the inheritance hierarchy for the class are also invoked, unrolling down the inheritance hierarchy from top to bottom. As each after method is executed the class-queries within the method are evaluated sequentially. In this manner, axiom verification for an entire flow diagram occurs simply by invoking the **check-axioms** method on all class instances. The fourth function of the model analyzer, information elicitation, uses one CLIPS function, containing a procedural algorithm, to represent each category of information to be elicited: timer periods, maximum rates, element cardinality, and model addenda.

The model analyzer implemented within CODA can be used to classify COBRA concepts on data flow/control flow diagrams, to verify that all concept instances are classified fully, to identify any concept instances that violate relevant axioms, and to elicit additional information

about concept instances. The next section illustrates the application of the model analyzer to a portion of a data flow/control flow diagram from a COBRA model for an automobile cruise control and monitoring system. The concurrent design that CODA generates from the COBRA model is also shown.

7. Case Study: Automobile Cruise-Control and Monitoring System

An automobile cruise-control and monitoring system, based on a COBRA model described by Gomaa, (Gomaa, 1993) serves as a case study for the application of knowledge-based graphical models, as implemented in the CODA model analyzer. The model analysis is discussed in Section 7.1. Section 7.2 illustrates the concurrent design generated by CODA, given the data flow/control flow model provided by Gomaa.

7.1 CODA's Analysis of the COBRA Model

Figure 10 shows the context diagram for the system. The context diagram is annotated with information inferred by CODA's model analyzer, or elicited from the designer. The annotations are shown on the context diagram, and on all subsequent data flow/control flow diagrams, enclosed within square brackets and set off in italicized print. Symbols, defined in Table 2, are used with each annotation to indicate the source of the information. To aid in the automatic classification of COBRA concepts, the context diagram depicted in Figure 10 supplements the original with event flows arriving from interrupt-driven, external devices. The presence of

Table 2. Symbols Used to Annotate Data Flow/control Flow Diagrams

Symbol	Meaning
#	This classification is directly represented using RTSA.
=	CODA inferred this classification.
?	CODA tentatively inferred this classification, but the designer was asked to confirm or override the classification.
+	CODA elicited additional information from the designer and then inferred this classification based on that additional information.
@	CODA elicited this classification from the designer.
*	CODA elicited this additional information from the designer after classifying the concept and determining that some required information was missing.

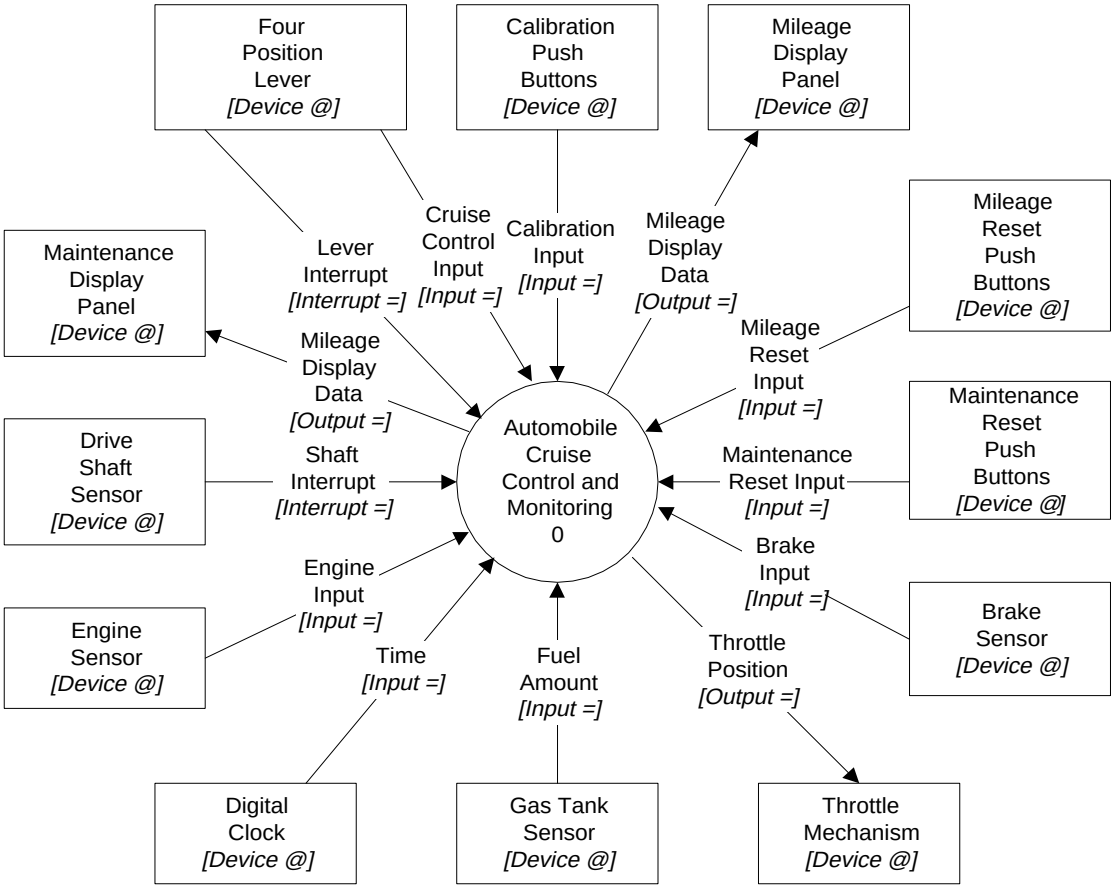

Figure 10. Annotated Context Diagram for Automobile Cruise Control and Monitoring System

interrupt-driven devices is described in the textual specification. By explicitly showing these events on the context diagram, CODA can make proper inferences from the terminators.

Data flow/control flow diagrams for most large systems are arranged hierarchically, as presented for example in Figures 11 and 12, to help human beings better comprehend the model. CODA, however, works from a flattened hierarchy of data flow/control flow diagrams. Each annotated element that appears on a context diagram or on a data flow/control flow diagram represents an element that cannot be further decomposed. For example, each terminator on the context diagram cannot be decomposed, and thus the terminators exist as part of the flattened hierarchy seen by CODA. Contrast these with the data transformation, Automobile Cruise Control and Monitoring. This data transformation is not annotated because it is decomposed on additional diagrams. The flattened data flow/control flow diagram for this case study consists of

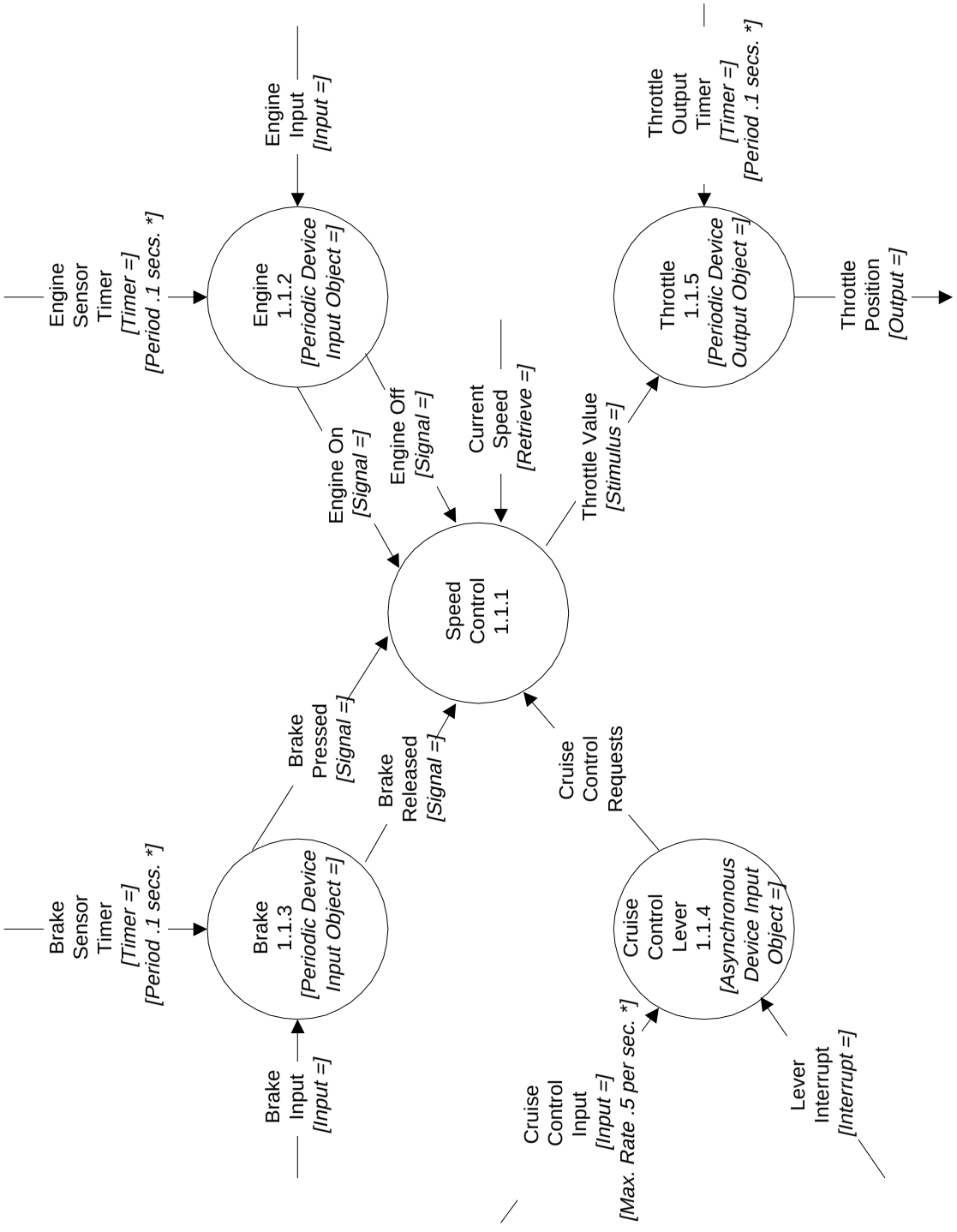

Figure 11. Fragment of an Annotated Data Flow/control Flow Diagram Containing a Decomposition of Automobile Control

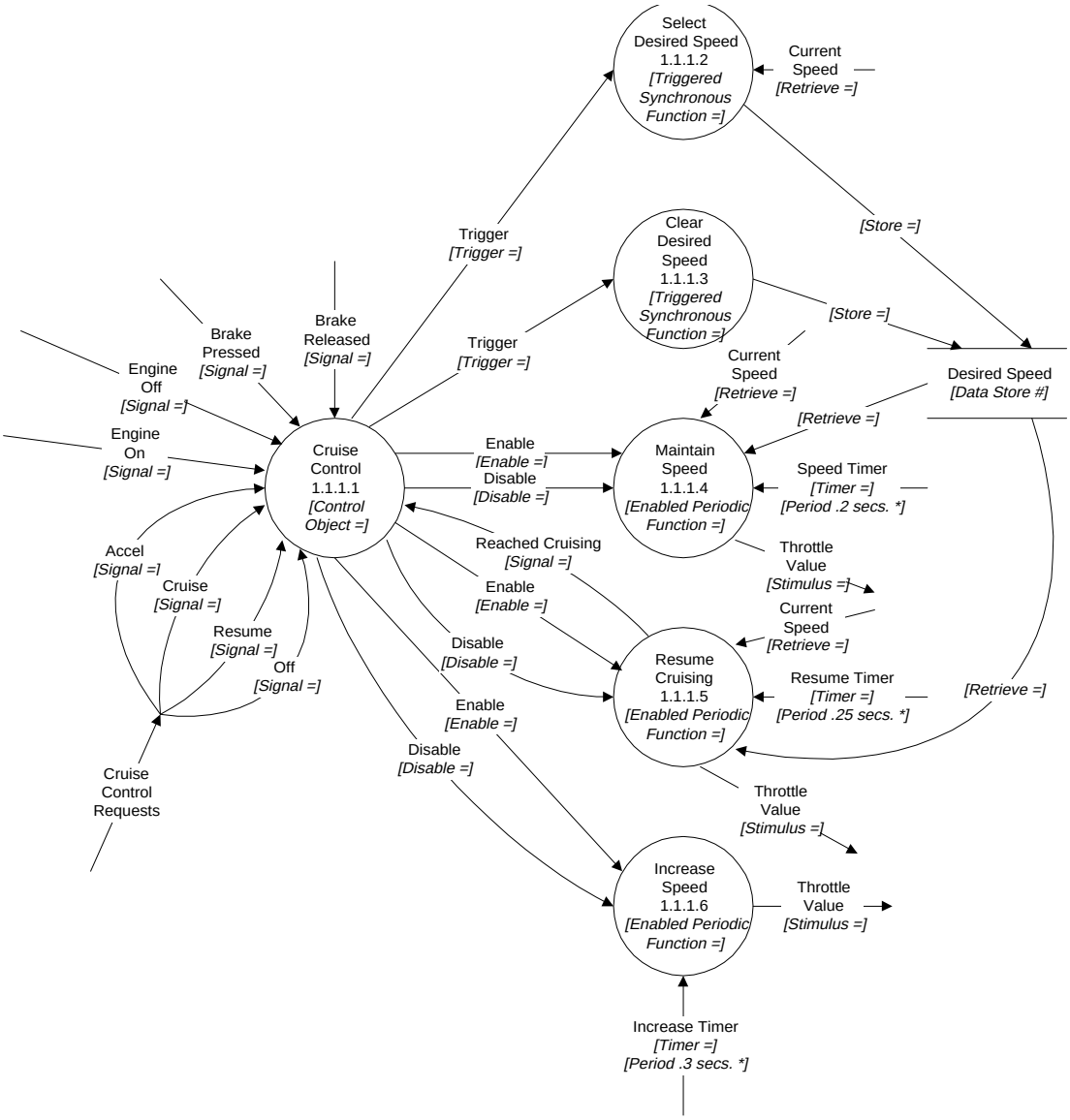

Figure 12. Decomposition of Speed Control

58 nodes (33 transformations, 12 terminators, and 13 data stores) and 112 arcs (69 data flows and 43 event/control flows).

Classifying concepts for this model requires a dialog between CODA and the designer; however, CODA can make most of the classification decisions without consulting the designer. At startup, CODA prompts for the designer's level of experience. When the designer is inexperienced CODA will make all decisions without consulting the designer. When the

designer is experienced, as in this case study, CODA consults the designer from time to time, where such consultation might prove advantageous. In this case study, only two consultations are needed. First, the designer, asked about the nature of the terminators in the model, indicates that all terminators are devices. Second, during the latter stages of classification, CODA discovers six data transformations that appear to be synchronous functions. Knowing the designer to be experienced, CODA presents each of these tentative classifications to the designer for confirmation.

After completing concept classification, CODA notes that the new semantic identities established for some of the model elements require the designer to supply additional information. For example, in this case study, sixteen event flows represent timers. CODA enables the designer to provide a positive period for each timer. After finishing the concept classification and information elicitation phases, CODA verifies that each element in the model is classified completely and that each element satisfies all relevant axioms.

To aid in understanding CODA's model analysis, a fragment of the data flow/control flow diagram for the automobile cruise-control and monitoring system is presented in Figures 11 and 12. A full treatment of this case study, including a complete, annotated data flow/control flow diagram and resulting concurrent design, is given elsewhere (Mills, 1996). To keep this paper brief, only the cruise-control subsystem is shown. Figure 12 decomposes Speed Control from Figure 11. To give the reader an idea about the inferences needed to classify concepts, several classification examples follow.

In Figure 11, the concept classifier determines the Brake object to be a periodic device input object because the Brake: (1) receives input from the brake sensor device and (2) is stimulated by the brake sensor timer. Referring back to Figure 8, the reader can follow the chain of rule firings that lead to this inference. In Figure 12, the concept classifier identifies the Maintain Speed function as an enabled periodic function because the function: (1) is enabled and disabled by a control object, (2) is stimulated by a timer and (3) receives inputs from no other transformations. In Figure 12, the concept classifier labels the Select Desired Speed function as a triggered synchronous function because the function: (1) is triggered by a control object, and (2) interacts only with data stores (and thus can complete its processing at the triggering state-transition).

7.2 CODA's Concurrent Design Generation

CODA's model analyzer provides a front-end to a concurrent design generator (Mills and Gomaa, 1996) that encodes heuristics from the CODARTS design method (Gomaa 1993). CODA's design generator implements the four steps used by CODARTS to transform a COBRA model into a concurrent design: (1) task structuring, (2) task interface definition, (3) module structuring, and (4) task and module integration. (Refer back to Section 4.2 for a description of these steps.) Figure 13 provides an overview of the design generated by CODA for the cruise-control subsystem discussed previously in Section 7.1. As an example of task structuring, CODA creates one periodic input task to poll both the Brake and Engine objects in Figure 11, because both are periodic device input objects with identical periods. As an example of task interface definition, CODA creates a message queue to hold asynchronous events sent from several other tasks to the Control Cruising task, because the destination task contains an embedded finite-state machine. As an example of module structuring, CODA forms a data abstraction module, Desired Speed, with three operations: Select, Clear, and Read. CODA generates this module from model elements shown in Figure 12: one data store, Desired Speed, two triggered synchronous functions, Clear Desired Speed and Select Desired Speed, and two retrieve data flows. As an example of task and module integration, CODA places the Desired Speed module outside any task, because the module is shared by the Control Cruising and Speed Control tasks. In addition, CODA designates that the Select and Clear operations are invoked by Control Cruising, while the Read operation is invoked by Speed Control.

8. Evaluation and Discussion

CODA's model analyzer was applied to four real-time systems modeled using RTSA notation. In addition to the Automobile Cruise-Control and Monitoring System discussed in the previous section, these systems included: a robot controller (Gomaa, 1993), an elevator control system (Gomaa, 1993), and a remote temperature sensor. (Nielsen and Shumate, 1987; Nielsen and Shumate, 1988)

8.1 Summary of Results

Table 3 presents an overview of the classification results across all case studies. Each element of a model can be depicted on a data flow/control flow diagram using one of seven RTSA syntactic components (see Figure 2). For conciseness, these syntactic elements are represented in five

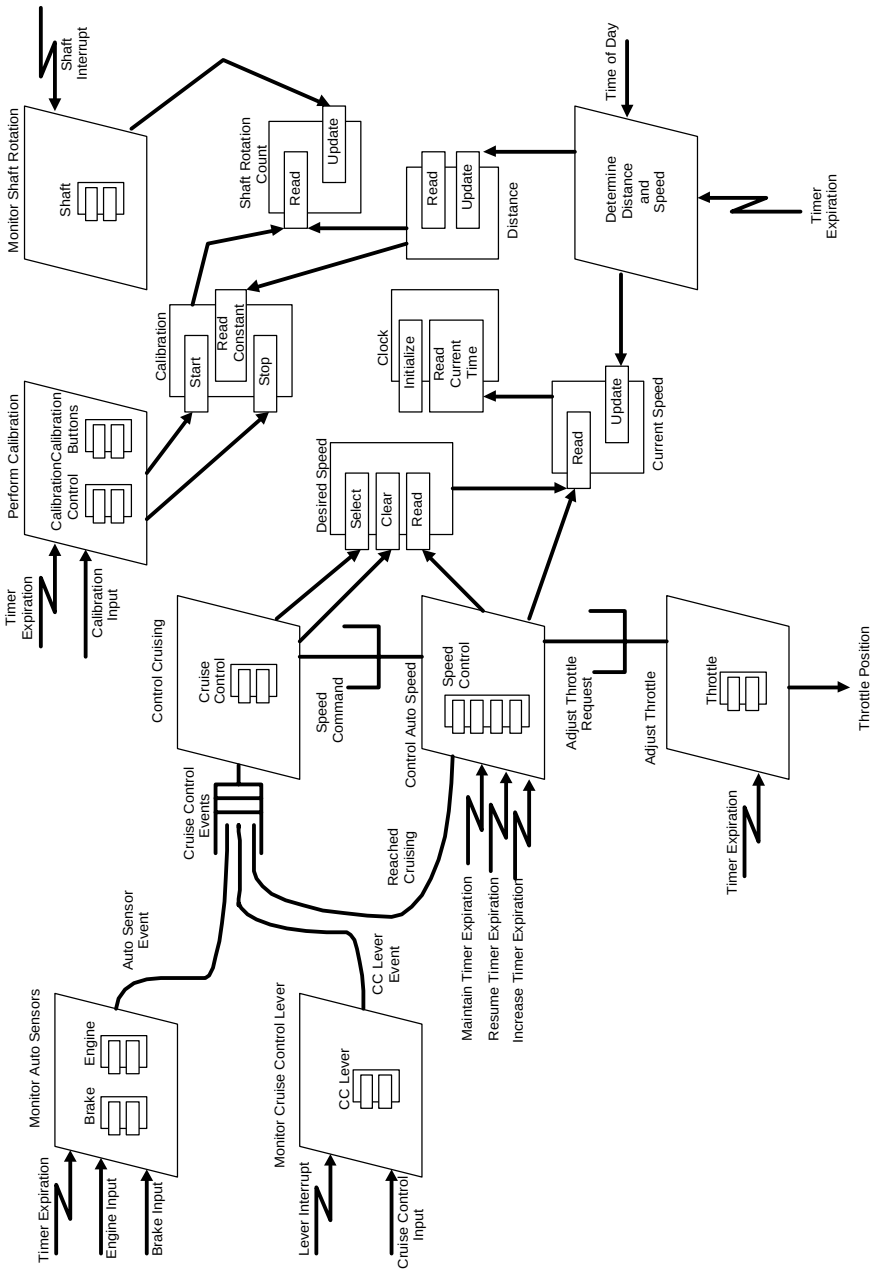

Figure 13. Concurrent Design Generated by CODA for the Cruise-Control Subsystem

rows within Table 3. Row two, Transformations, includes both data and control transformations; row seven, Data Flows, includes both unidirectional and bi-directional data flows. Row one of Table 3 represents all nodes on all flow diagrams, row five represents all arcs, and row eight

represents all elements (that is, nodes plus arcs). The first column in Table 3 simply gives the total number of each type of element that appears on the data flow/control flow diagrams in the case studies; specifically, the case-study diagrams contained 358 elements as follows: (1) 125 nodes of which 79 were transformations, 28 were terminators, and 18 were data stores and (2) 233 arcs of which 91 were event flows and 142 were data flows. The remaining five columns in Table 3 represent the manner in which classification is achieved for the elements represented by each intersecting row.

Table 3. Classifications Over All Models for the Case Studies

	Total	Directly Represented	Inferred Automatically	Inferred Tentatively	Required Information	Designer Classified
All Nodes	125	18 (14%)	59 (47%)	8 (6%)	12 (10%)	28(22%)
Transformations	79	0 (0%)	59 (75%)	8 (10%)	12 (15%)	0 (0%)
Terminators	28	0 (0%)	0 (0%)	0 (0%)	0 (0%)	28 (100%)
Data Stores	18	18 (100%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
All Arcs	233	0 (0%)	231 (99%)	0 (0%)	2 (1%)	0 (0%)
Event Flows	91	0 (0%)	91 (100%)	0 (0%)	0 (0%)	0 (0%)
Data Flows	142	0 (0%)	140 (99%)	0 (0%)	2 (1%)	0 (0%)
All Elements	358	18 (5%)	290 (81%)	8 (2%)	14 (4%)	28 (8%)

A review of the last row of Table 3 indicates the degree of success achieved when CODA's model analyzer, using the classification rules described in Section 5.3.1, is employed against the data flow/control flow diagrams for the four case studies. As shown in the two columns in Table 3 labeled Directly Represented and Inferred Automatically, CODA succeeded without help in classifying about 86% of the elements in the data flow/control flow diagrams.

Where no human assistance is available, CODA makes default decisions that would have proven accurate in all but one of the 358 classification decisions comprising the four case studies reported in this paper. CODA achieves this success by invoking default rules that take reasonable design decisions in the absence of additional guidance. This ability to generate effective designs without human intervention allows CODA's design generator to be used in

situations where human assistance is unavailable or unwanted. For the case studies reported in this paper, human assistance was available and provided.

As shown in the last column of Table 3, the designer provided the classification for terminators, 8% of the elements in the case studies. Cases worth considering in detail involve those 2% where CODA's classification was inferred tentatively but referred to the designer for confirmation and those 4% where CODA's classification occurred only after the designer supplied additional information. In these cases, a large difference exists between the classification of arcs, where 99% are classified without help, and the classification of transformations, where only 75% are classified without help.

8.2 Problems Classifying Data Flows

Only two data flows, from among the 233 arcs considered in the case studies, cannot be classified without consulting the designer. Both of these appear in the same data flow/control flow diagram; in fact, the two data flows are related. In this situation two data transformations exchange data flows with each other. CODA cannot determine which of these two data flows, if either, is sent in response to the other; therefore, the designer must be consulted. The designer should know, or be able to determine from the pseudo-code associated with each data transformation, whether one of the data flows is sent in response to the other. Of course, the designer might not know this information, so CODA is prepared to make a default decision. Only in situations such as this one will CODA's automated classifier need to consult the designer to classify arcs on a flow diagram. Thus, the performance of the automated classifier against arcs will depend on the number of these cases that exist in a given diagram. The performance of CODA's automated classifier against transformations appears less effective.

8.3 Problems Classifying Transformations

Table 4 provides a breakdown of transformation classifications by case study. For the four case studies, CODA could classify definitively 75% of the transformations, could classify tentatively 10%, and could classify the remaining 15% only after hints by the designer. All control objects are inferred automatically from their syntax.

CODA's performance is much better in three of the case studies: automobile cruise control, robot controller, and elevator control system. Considering only these three cases, 83% (54/65) of transformations were classified automatically, 11% (7/65) could be classified tentatively, while

Table 4. Classification of Transformations by Case Study

Case Study	Transformations	Inferred Automatically	Inferred Tentatively	Required Information
All Models	79	59 (75%)	8 (10%)	12 (15%)
Automobile Cruise Control	33	27 (82%)	6 (18%)	0 (0%)
Robot Controller	18	14 (78%)	1 (5%)	3 (17%)
Elevator Control System	14	13 (93%)	0 (0%)	1 (7%)
Remote Temperature Sensor	14	5 (36%)	1 (7%)	8 (57%)

only 6% (4/65) required hints in order to make a classification. For the remote temperature sensor, CODA's performance is singularly poor. This dichotomy exists because three of the models were developed using object-based decomposition, as recommended in COBRA, while the remote temperature sensor specification was developed using functional decomposition, as recommended in Structured Analysis (DeMarco, 1978). In the data flow diagram for the remote temperature sensor, as provided by Nielsen and Shumate, (Nielsen and Shumate, 1987; Nielsen and Shumate, 1988) numerous aperiodic functions defy classification without assistance from the designer. A detailed analysis of CODA's tentative and assisted classification for transformations follows.

8.3.1 Tentative Classifications. Each of the eight transformations that CODA classified tentatively involve the same type of situation. Whenever CODA encounters a function on a data flow/control flow diagram such that the function sends data only to data stores or passive device-interface objects, or to both, then, if the function is not classified otherwise, CODA tentatively identifies the function as synchronous. This tentative classification assumes that, for real-time systems, updating data stores and writing to passive devices is generally a fast operation that can be completed atomically. This assumption is usually correct. In some situations, however, an operation might take long enough that the designer chooses to view the function as asynchronous. In the particular case studies covered in this paper for example, the designer confirms the tentative classification in seven of the eight cases. In one case, the designer overrides CODA's tentative classification where a function updates a data store. The designer overrides CODA based on application-specific knowledge that the data store is large enough, or

that the update algorithm is time-consuming enough, to warrant asynchronous processing. This information is not available to CODA but might be available to the designer. When the designer does not know whether to override CODA's tentative decision, then the decision stands.

8.3.2 Assisted Classifications. In some situations CODA recognizes that additional information might be available that can help make a more accurate classification. In these situations, represented in the last column of Table 4, CODA consults the designer to see what other information exists. Lacking additional information, CODA makes a default classification. Table 4 indicates twelve instances among the case studies where the designer is consulted to help classify a transformation. These twelve instances represent two general situations. The first situation occurs when a function is triggered by a control transformation, yet the triggered function receives input data from some other transformation. In situations of this type, CODA recognizes that the triggered function might not be able to execute at the triggering state-transition because the input data might be unavailable. The designer is consulted on this question and CODA then makes the best classification based upon any additional information provided. The second, and more frequent, situation occurs when a function receives input from only a single source, or the same input from multiple sources. In such situations, CODA recognizes that the function might be classified as either a synchronous or asynchronous function depending upon the execution time required. After consulting the designer, CODA makes the best classification based on the information available. Had no human assistance been available, CODA's default decisions would have proven accurate in all but one of the 79 transformation classifications reported in Table 4.

8.4 Automated Design Generation

In large part, the effectiveness of CODA's model analyzer can be evaluated based on the results produced by CODA's design generator. CODA's design generator was used to automatically generate ten, distinct, concurrent designs for the preceding four real-time systems, modeled as data flow/control flow diagrams (Mills, 1996). Multiple designs were generated for each system to test the ability of CODA's design generator to adapt to variations in the intended target environment. Of the 1,568 CODARTS design decisions required to generate the ten designs, 1,524, or 97%, were taken without human intervention. The effectiveness of CODA's design generator derives in large measure from CODA's model analyzer, which creates the semantic

view needed to effectively apply the CODARTS heuristics encoded within CODA's design generator.

9. Conclusions

This paper proposed an approach, called knowledge-based graphical models, capable of inferring the presence of semantic concepts from a graphical model. To illustrate the approach, the paper specified a knowledge-based graphical model for a specific modeling method, COBRA, and then described a means of implementing the approach as a model analyzer embedded within CODA, a concurrent designer's assistant. The approach was evaluated by applying CODA to analyze data flow/control flow models for four real-time systems. For the four systems, CODA inferred, automatically and correctly, the existence of 86% of all semantic concepts within the flow diagrams. Varying degrees of human assistance were used to correctly identify the remaining semantic concepts within the diagrams: in two percent of the cases CODA reached tentative classifications that a designer was asked to confirm or override; in four percent of the cases a designer was asked to provide additional information; in the remaining eight percent of the cases, all involving terminators, the designer was asked to identify the semantic concept. When providing assistance, a human designer consults the textual description accompanying the graphical model for each system. Using the semantic interpretation provided by CODA's model analyzer, CODA's design generator proved very effective. During the generation of ten distinct designs for the four case studies, CODA's design generator made 97% of all design decisions without consultation.

The approach described in this paper can be applied to assist designers in the creation of concurrent designs. A tool such as CODA could be embedded as a component in a computer-aided software engineering (CASE) tool. Most CASE tools enable a designer to enter flow diagrams and structure charts, or other representations of a software design; however, the process of creating the software design from the flow diagrams must be performed by a human designer, outside the CASE tool and without automated assistance. Where a component such as CODA is available, a designer could enter a data flow/control flow diagram into a CASE tool and then invoke automated assistance to generate a design. Such automation can capture design decisions and rationale and can maintain traceability between elements on the flow diagram and components in the design.

Knowledge-based graphical models should be applicable to a range of software design methods that model a system using graphical notations and accompanying textual descriptions. For example, the semantic concepts discussed in this paper appear quite similar in intent to stereotypes within the Unified Modeling Language (UML) (Fowler, 1997). A taxonomy of UML stereotypes, including axioms, can be devised in a form similar to that depicted in this paper for the COBRA concept taxonomy. Rules could then be formulated for attempting to classify UML model elements against the taxonomy of stereotypes. In addition, UML model elements labeled with stereotypes by a designer could be evaluated automatically against a relevant hierarchy of axioms. Model elements with associated stereotypes could also trigger the automatic elicitation of information required to support later phases of the software design process. Once a UML model is properly labeled with stereotypes and augmented with additional information, the CODARTS design heuristics encoded within CODA's design generator could be used to produce a concurrent design from a UML model.

References

Boloix, G., Sorenson, P. G., and Tremblay, J. P. (1992) "Transformations using a meta-system approach to software development", *Software Engineering Journal*, November 1992, pp. 425-437.

Booch, G. (1986) "Object-Oriented Development", *IEEE Transactions on Software* Engineering, February 1986, pp. 211-221.

Booch, G. (1991) Object Oriented Design With Applications, Benjamin/Cummings, Redwood City, California, 1991.

Brachman, R. J. and Schmolze, J. G. (1989), "An Overview of the KL-ONE Knowledge Representation System", in Readings in Artificial Intelligence and Databases, (J. Mylopoulos and M. L. Brodie, eds.), Morgan Kaufmann Publishers, Inc., San Francisco, CA. 1989. pp. 207-229..

Coad, P. and Yourdon, E. (1991) *Object-Oriented Analysis*, Yourdon Press, Englewood Cliffs, New Jersey, 1991.

DeMarco, T. (1978) Structured Analysis and System Specification, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

Fikes, R. and Kehler, T. (1985) "The Role of Frame-Based Representation In Reasoning", *Communications of the ACM*, September 1985, Volume 28, Number 9, pp. 904-920.

Fowler, M. (1997) UML Distilled, (with Kendall Scott), Addison-Wesley, 1997.

Genesereth, M. R. and Ginsberg, M. L. (1985) "Logic Programming", *Communications of the ACM*, September 1985, Volume 28, Number 9, pp. 933-941.

Gomaa, H. (1993) Software Design Methods for Concurrent and Real-Time Systems, Addision-Wesley Publishing Company, Reading Massachusetts, 1993.

Hatley, D. and Pirbhai, I. Strategies for Real-time System Specification, Dorset House, New York, 1988.

Hayes-Roth, F. (1985) "Ruled-Based Systems", *Communications of the ACM*, September 1985, Volume 28, Number 9, pp. 921-932.

Jackson, M. (1983) System Development, Prentice-Hall, Englewood Cliffs, NJ, 1983.

Karimi, J. and Konsynski, B. R. (1988) "An Automated Software Design Assistant", *IEEE Transactions on Software Engineering*, February 1988, pp. 194-210.

Lim, E. and Cherkassky, V. (1992) "Semantic Networks and Associative Databases", *IEEE Expert*, August 1992, pp. 31-40.

Lor, K. E. and Berry, D. M. (1991) "Automatic Synthesis of SARA Design Models From Systems Requirements", *IEEE Transactions on Software Engineering*, December 1991, pp. 1229-1240.

Michalski, R. S. (1980), "Pattern Recognition as Rule-Guided Inductive Inference", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Volume 2, Number 4, 1980.

Mellor, S. J. and Ward, P. T. (1986) Structured Development for Real-Time Systems, Vol. 3: Implementation Modeling Techniques, Yourdon Press, Englewood Cliffs, NJ, 1986.

Mills, K. (1996) Automated Generation of Concurrent Designs For Real-Time Software, Ph.D. Dissertation, George Mason University, 1996.

Mills, K and Gomaa, H. (1996) "A Knowledge-based Approach for Automating a Design Method for Concurrent and Real-Time Systems", *Proceedings of the 8th International Conference on Software Engineering and Knowledge Engineering*, June 10-12, 1996.

NASA (National Aeronautics and Space Administration). (1993) Software Technology Branch, CLIPS Reference Manual, Three Volumes, CLIPS Version 6.0, June 2, 1993.

Nielsen, K. and Shumate, K. (1987) "Designing Large Real-Time Systems With Ada", *Communications of the ACM*, August 1987, pp. 695-715.

Nielsen, K. and Shumate, K. (1988) Designing Large Real-Time Systems With Ada, McGraw-Hill, New York, 1988.

Rumbaugh, J.; Blaha, M.; Premerlani, W.; Eddy, F.; and Lorensen, W. (1991) Object-Oriented Modeling and Design, Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

Shlaer, S. and Mellor, S. J. (1992) Object Lifecycles - Modeling the World in States, Yourdon Press, Englewood Cliffs, New Jersey, 1992.

Tsai, J. P. and Ridge, J. C. (1988) "Intelligent Support for Specifications Transformation", *IEEE Software*, November 1988, pp. 28-35.

Ward, P. and Mellor, S. (1985) Structured Development for Real-time Systems, Four Volumes, Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

Yourdon, E. and Constantine, L.L. (1979) Structured Design, Prentice-Hall, Englewood Cliffs, New Jersey, 1979.